Московский авиационный институт

(национальный исследовательский университет)

Институт №8 "Информационные технологии и прикладная математика"

**Курсовой проект по курсу**

**"Инструментальные средства разработки прикладных программных систем»**

**по теме "Алгоритм преобразования произвольного графа G в лес, состоящий из самоподобных изоморфных графов"**

Выполнил студент группы:

М8О-205Б-21

Глушин Никита Дмитриевич

Преподаватель:

Семенов Александр Сергеевич

Оценка: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Дата:  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Тема 4.6.** Алгоритм преобразования произвольного графа G в лес, состоящий из самоподобных изоморфных графов.

**Постановка задачи и описание алгоритма**

Разработать алгоритмы разделения произвольного графа на подграфы, используя заданные шаблоны фрактальных графов: гиперкуб, решетка, бинарное дерево, линия.

**Гиперкуб.**

Идея заключается в переборе подграфов графа из 8 вершин, начиная с вершин с большей степенью. Каждый подграф проверяется на то, является ли он гиперкубом. Для этого каждая из его вершин должна иметь степень, не меньшую трех, а также они должны быть соседями друг друга. Затем все найденные гиперкубы объединяются.

При этом в памяти компьютера хранятся только N вершин, где N - количество вершин заданного графа, а в реализации алгоритма перебираются указатели на них.

**Решетка.**

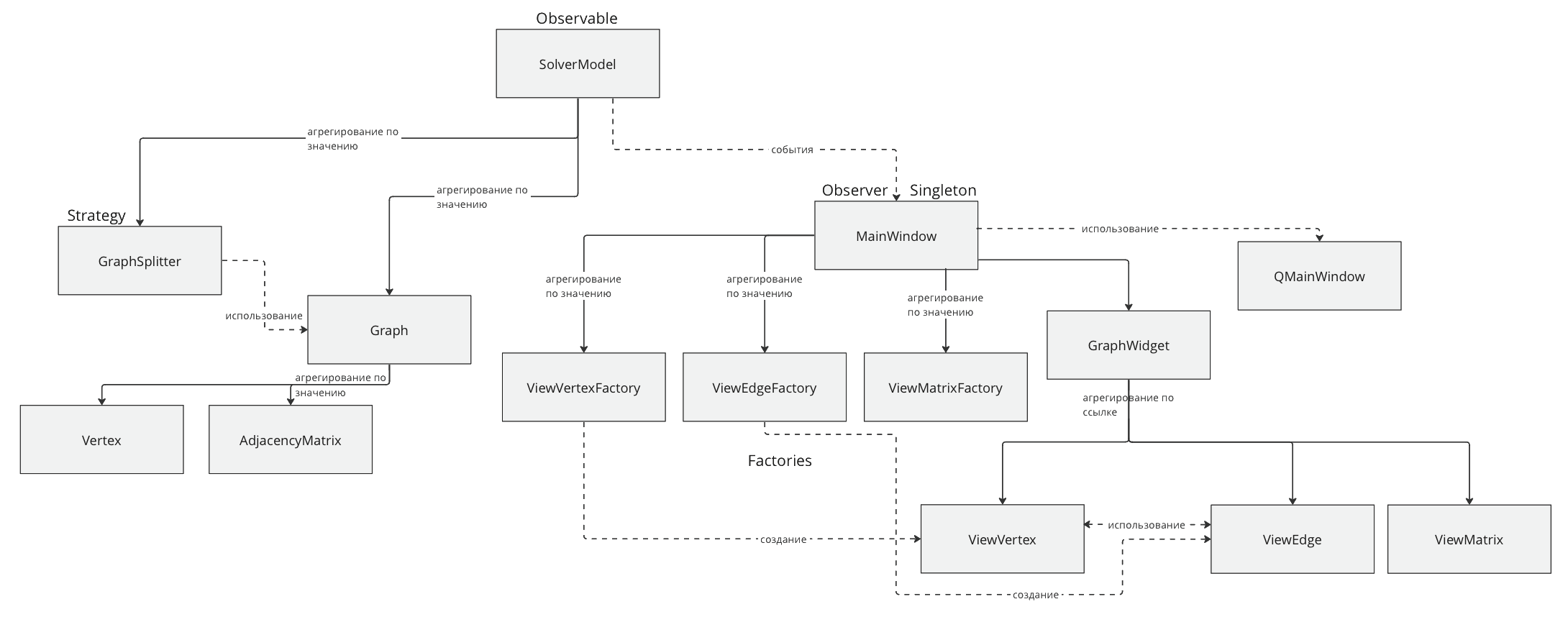
Алгоритм схож с нахождением гиперкуба. Аналогично перебираются подграфы, состоящие из четыре вершин. Каждая из них должна иметь степень, не меньшую двух, а вместе они должны образовывать элементарную решетку. Результатом алгоритма будет их объединение.

**Бинарное дерево.**

Найдем сначала все возможные «корни» бинарных деревьев, т.е. вершины со степенью, большей или равной 2. Затем будем перебирать их и соединять друг с другом те, которые имеют общее ребро в заданном графе, и, степень которых в новом графе меньше двух. Следующим шагом соединим корни с листьями, где это возможно. В завершение обойдем новый граф и постараемся соединить вершины, степень которых меньше двух.

**Линия.**

Алгоритм нахождения линий достаточно прост. Для каждой вершины продвигаемся «вглубь» графа и подсоединяем соседа.

**Диаграмма классов**

**Реализация алгоритмов на C++**

**Гиперкуб.**

**void** GraphSplitter::getCubeFromVertexIfExists(Vertex\* vertex, Graph &res, std::vector<std::pair<**int**, **int**> > &levels) {

**auto** neighbours = vertex->Neighbours();

**if** (neighbours.size() < 3) **return**;

**for** (Vertex\* v1 : neighbours) { // перебираем первые 3 вершины

**for** (Vertex\* v2 : neighbours) {

**for** (Vertex\* v3 : neighbours) {

**if** (v1 != v2 && v1 != v3 && v2 != v3 && v1 != vertex && v2 != vertex && v3 != vertex) { // если все уникальные

**auto** NeighboursV1 = v1->Neighbours();

**auto** NeighboursV2 = v2->Neighbours();

**auto** NeighboursV3 = v3->Neighbours();

**auto** commonNeighboursV1V2 = getCommonNeighbours(NeighboursV1, // находим общих соседей

NeighboursV2);

**auto** commonNeighboursV1V3 = getCommonNeighbours(NeighboursV1,

NeighboursV3);

**auto** commonNeighboursV2V3 = getCommonNeighbours(NeighboursV2,

NeighboursV3);

**if** (commonNeighboursV1V2.empty() || commonNeighboursV1V3.empty() || commonNeighboursV2V3.empty())

**continue**;

**for** (Vertex\* v4 : commonNeighboursV1V2) { // перебираем следующие 3 вершины среди общих

**for** (Vertex\* v5 : commonNeighboursV1V3) {

**for** (Vertex\* v6 : commonNeighboursV2V3) {

**if** (v4 != vertex && v5 != vertex && v6 != vertex) {

**auto** NeighboursV4 = v4->Neighbours();

**auto** NeighboursV5 = v5->Neighbours();

**auto** NeighboursV6 = v6->Neighbours();

**auto** commonNeighboursV5V6 = getCommonNeighbours(NeighboursV5,

NeighboursV6);

**auto** commonNeighboursV4V5V6 = getCommonNeighbours(

NeighboursV4,

commonNeighboursV5V6);

**if** (!commonNeighboursV4V5V6.empty()) { // получаем последнюю вершину

Vertex\* v7 = **nullptr**;

**for** (Vertex\* v : commonNeighboursV4V5V6) {

**if** (v != vertex &&

v != v1 &&

v != v2 &&

v != v3 &&

v != v4 &&

v != v5 &&

v != v6)

v7 = v;

}

**if** (!v7) **continue**;

addNewCube(**{**vertex, v1, v2, v3, v4, v5, v6, v7**}**, res, levels); // добавляем

}

}

}

}

}

}

}

}

}

}

**Решетка.**

void GraphSplitter::getGridFromVertexIfExists(Vertex\* vertex, Graph &res, std::vector<std::pair<int, int> > &levels) {

auto neighbours = vertex->Neighbours();

if (neighbours.size() < 2) return;

for (Vertex\* v1 : neighbours) { // перебираем первые 2 вершины

for (Vertex\* v2 : neighbours) {

if (v1 != v2) {

auto neighbours\_v1 = v1->Neighbours();

auto neighbour\_v2 = v2->Neighbours();

auto common\_neighbours = getCommonNeighbours(neighbours\_v1, neighbour\_v2); // общие соседи

if (!common\_neighbours.empty()) {

Vertex\* v3 = nullptr;

for (Vertex\* v : common\_neighbours) // выбираем общего соседа

if (v != vertex && v != v1 && v != v2) v3 = v;

if (!v3) continue;

if (v1 != v3 && v2 != v3 && v1 != vertex && v2 != vertex && v3 != vertex) { // если все уникальные

int cnt = res.HasVertex(vertex->Id()) + res.HasVertex(v1->Id()) + res.HasVertex(v2->Id()) +

res.HasVertex(v3->Id());

if (cnt == 0 || cnt == 2 || (cnt == 3 && !res.HasVertex(vertex->Id()) && // проверка на правильную решетку

(res.IsConnected(v1->Id(), v2->Id()) + res.IsConnected(

v1->Id(),

v3->Id())

+ res.IsConnected(v2->Id(), v3->Id())) == 2) &&

res.GetVertexById(v1->Id())->Degree() + 1 <= 3 &&

res.GetVertexById(v2->Id())->Degree() + 1 <= 3) {

addNewGrid({vertex, v1, v2, v3}, res, levels); // добавление

}

}

}

}

}

}

}

**Бинарное дерево.**

Graph GraphSplitter::buildGraph(const Vertex::VertexesOrderedByDegree& roots, std::vector<std::pair<int, int> > &levels) {

Graph res;

std::map<unsigned int, int> neighbours\_count;

std::set<unsigned int> wasAddedAsNeighbour;

for (Vertex\* root : roots) {

res.AddVertex(root->Id());

int countOfNeighbours = 0;

for (Vertex\* neighbour : root->Neighbours()) {

if (countOfNeighbours >= 2) break;

if (roots.count(neighbour) != 0 && !res.IsConnected(root->Id(), neighbour->Id()) // добавляем соседей-корни

&& neighbours\_count[neighbour->Id()] + 1 <= 2 && wasAddedAsNeighbour.count(neighbour->Id()) == 0) {

countOfNeighbours++;

res.AddVertex(neighbour->Id());

res.AddEdge(root->Id(), neighbour->Id());

levels.push\_back({root->Id(), neighbour->Id()});

neighbours\_count[root->Id()]++;

neighbours\_count[neighbour->Id()]++;

wasAddedAsNeighbour.insert(neighbour->Id());

}

}

for (Vertex\* neighbour : root->Neighbours()) {

if (countOfNeighbours >= 2) break;

if (roots.count(neighbour) == 0 && !res.IsConnected(root->Id(), neighbour->Id()) // добавляем соседей-листья

&& neighbours\_count[neighbour->Id()] + 1 <= 2 && wasAddedAsNeighbour.count(neighbour->Id()) == 0) {

countOfNeighbours++;

res.AddVertex(neighbour->Id());

res.AddEdge(root->Id(), neighbour->Id());

levels.push\_back({root->Id(), neighbour->Id()});

neighbours\_count[root->Id()]++;

neighbours\_count[neighbour->Id()]++;

wasAddedAsNeighbour.insert(neighbour->Id());

}

}

}

for (auto root : roots) {

if (res.GetVertexById(root->Id())->Neighbours().empty()) {

res.RemoveVertex(root->Id());

}

else if (res.GetVertexById(root->Id())->Degree() <= 2) { // объединяем разрозненные деревья в одно

for (auto vertex : root->Neighbours()) {

std::map<int, int> color;

bool was = res.HasVertex(vertex->Id());

bool was\_connected = res.IsConnected(root->Id(), vertex->Id());

res.AddVertex(vertex->Id());

res.AddEdge(root->Id(), vertex->Id());

levels.push\_back({root->Id(), vertex->Id()});

if (!was\_connected &&

res.GetVertexById(vertex->Id())->Degree() + 1 <= 3 && !hasCycle(root->Id(), -1, res, color)) {

break;

} else {

if (!was\_connected) {

res.RemoveEdge(root->Id(), vertex->Id());

levels.erase(std::remove(levels.begin(),

levels.end(),

std::make\_pair((int) root->Id(),

(int) vertex->Id())),

levels.end());

}

if (!was) res.RemoveVertex(vertex->Id());

}

}

}

}

return res;

}

**Линия.**

**bool** dfs(Vertex\* vertex, Graph &res, std::set<Vertex\*> &used, std::vector<std::pair<**int**, **int**> > &levels) {

**if** (used.count(vertex) > 0) **return false**;

used.insert(vertex);

**for**(Vertex\* neighbour : vertex->Neighbours()) {

**if** (used.count(neighbour) == 0) {

res.AddVertex(neighbour->Id());

res.AddEdge(vertex->Id(), neighbour->Id());

levels.push\_back({vertex->Id(), neighbour->Id()});

dfs(neighbour, res, used, levels);

**return true**;

}

}

**return false**;

}

Graph GraphSplitter::GetAllLines(Graph \*graph, std::vector<std::pair<**int**, **int**> > &levels) {

std::set<Vertex\*> used;

Graph res;

std::set<Vertex\*, cmp> vertexes;

**for** (**auto** vertex : graph->GetVertexes()) vertexes.insert(vertex);

**for** (Vertex\* vertex : vertexes) {

**if** (used.count(vertex) == 0) {

res.AddVertex(vertex->Id());

**if** (!dfs(vertex, res, used, levels)) { // погружаемся внутрь графа

res.RemoveVertex(vertex->Id());

used.erase(vertex);

}

}

}

**for** (Vertex\* vertex : vertexes) { // объединяем полученные линии

**for** (Vertex\* neighbour : vertex->Neighbours()) {

**if** (vertex->Degree() <= 1 && neighbour->Degree() <= 1) {

res.AddVertex(vertex->Id());

res.AddVertex(neighbour->Id());

res.AddEdge(vertex->Id(), neighbour->Id());

levels.push\_back({vertex->Id(), neighbour->Id()});

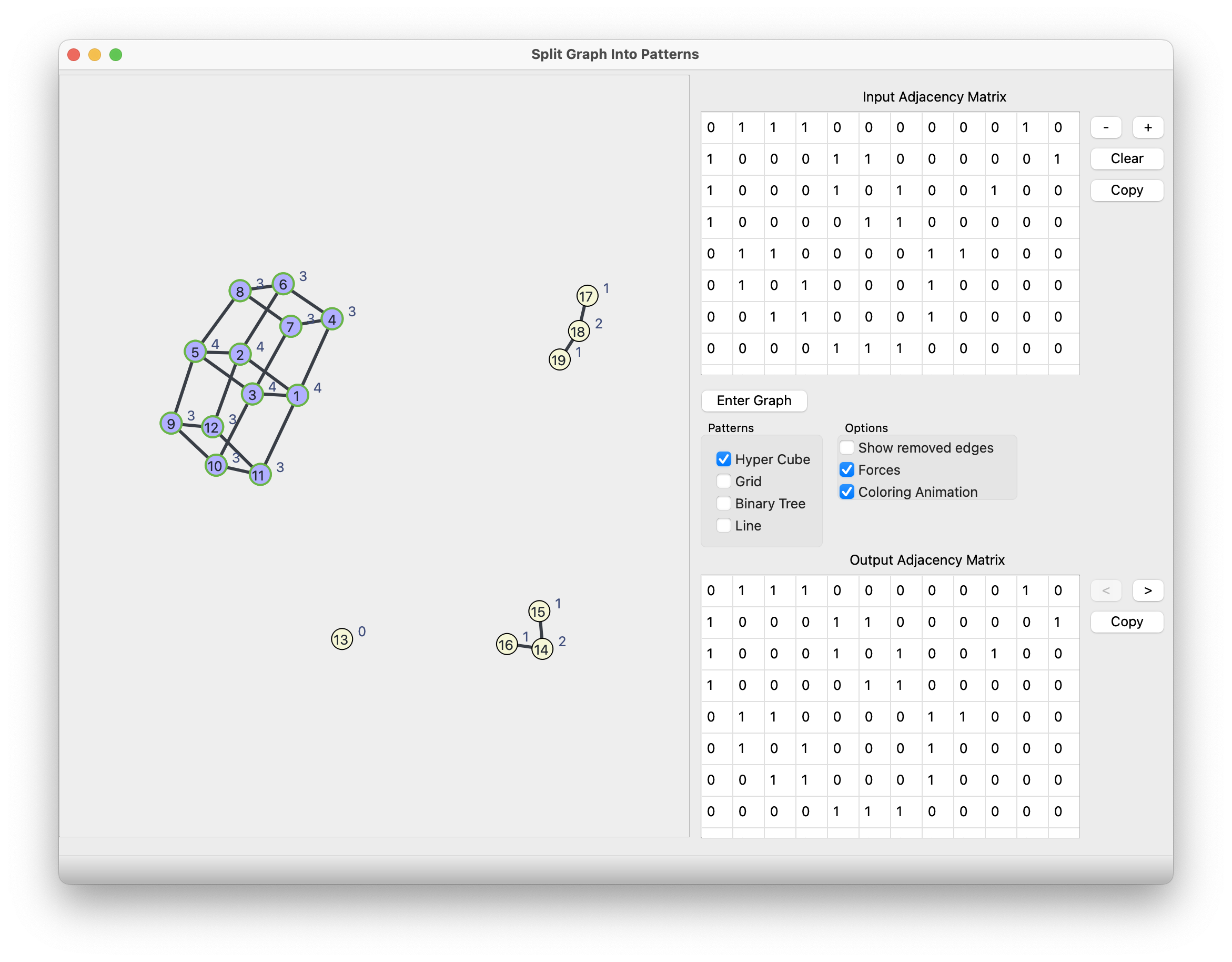
}

}

}

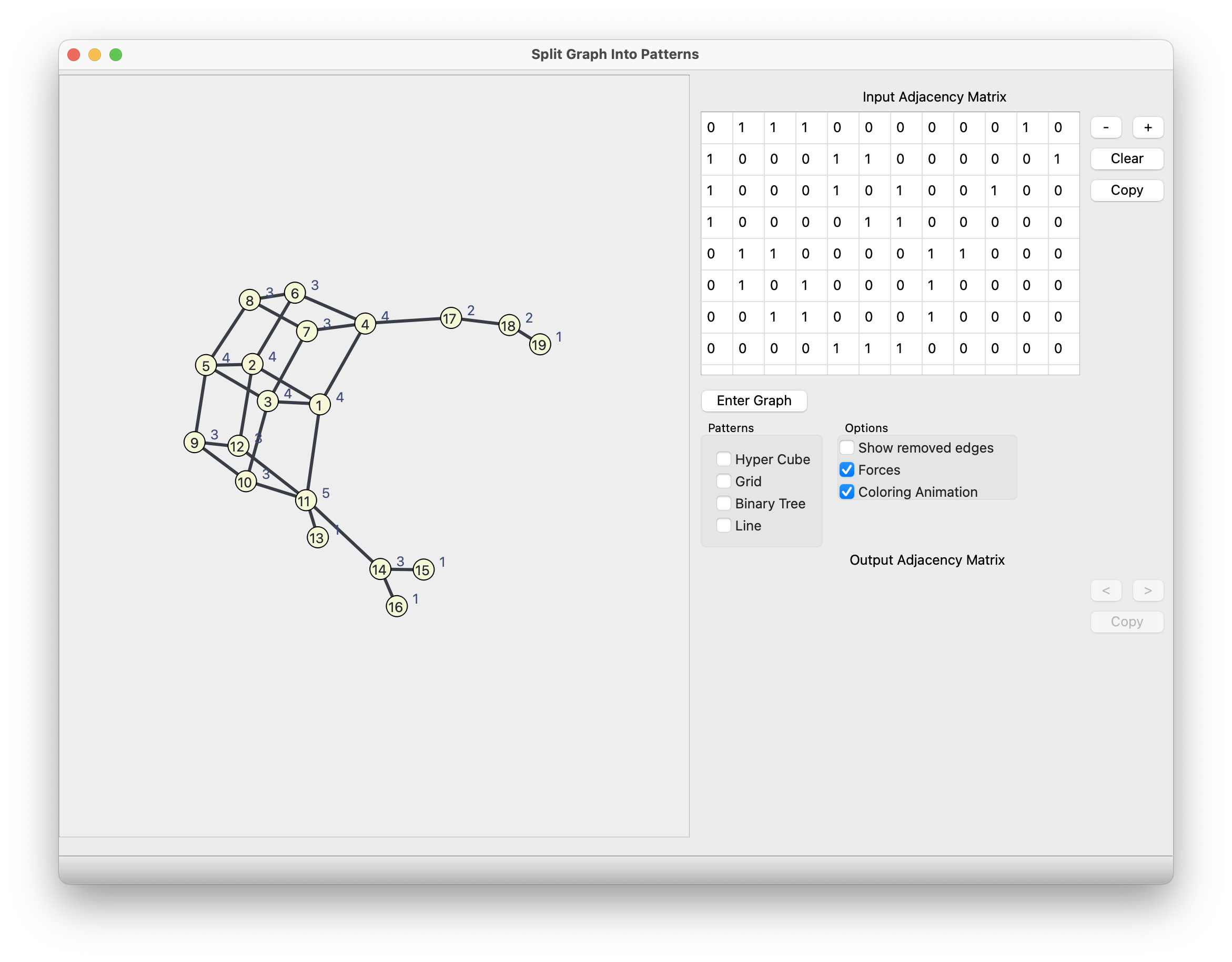
**return** res;

}

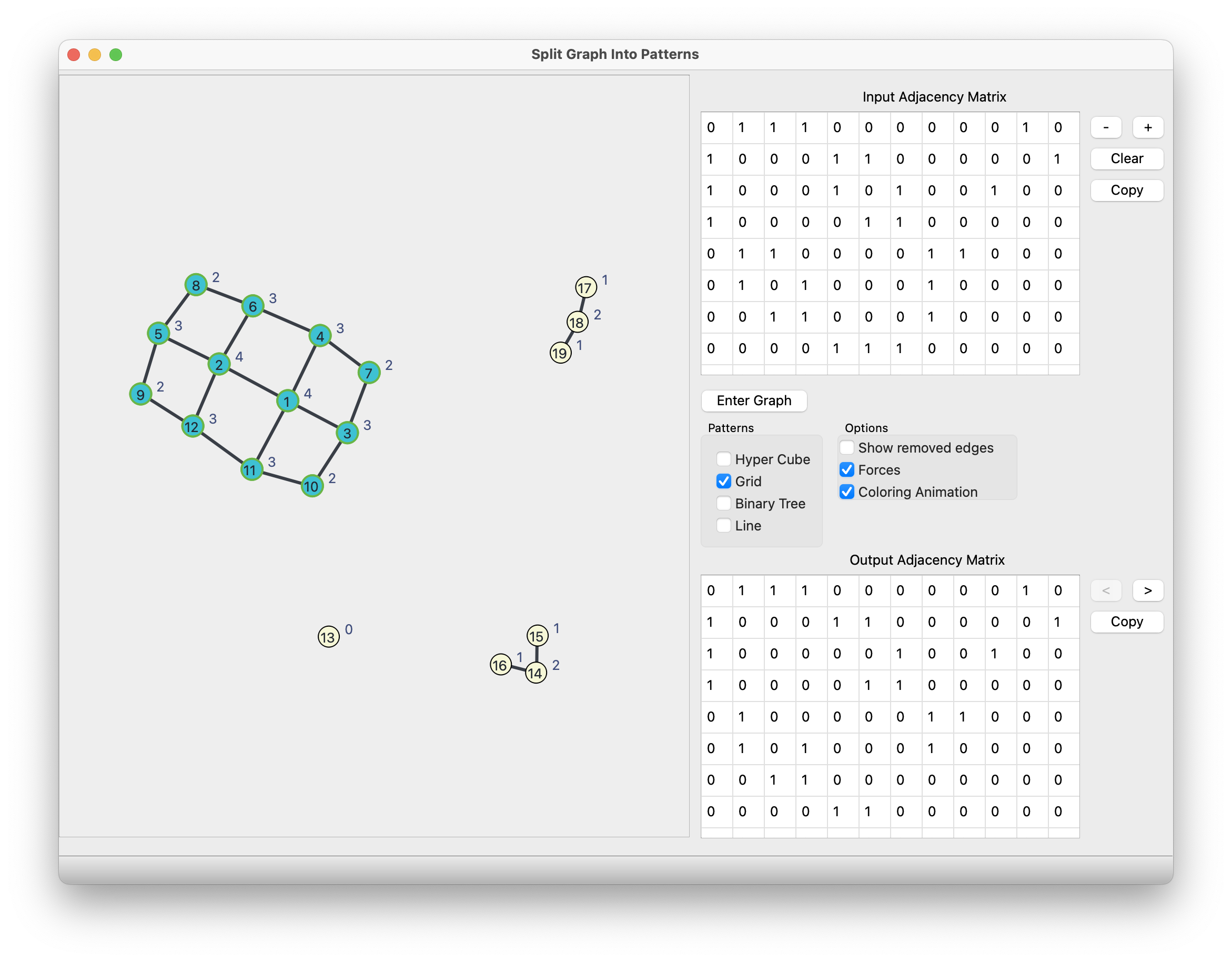
**Демонстрация функциональных возможностей приложения**

**Тесты. Протокол работы.**

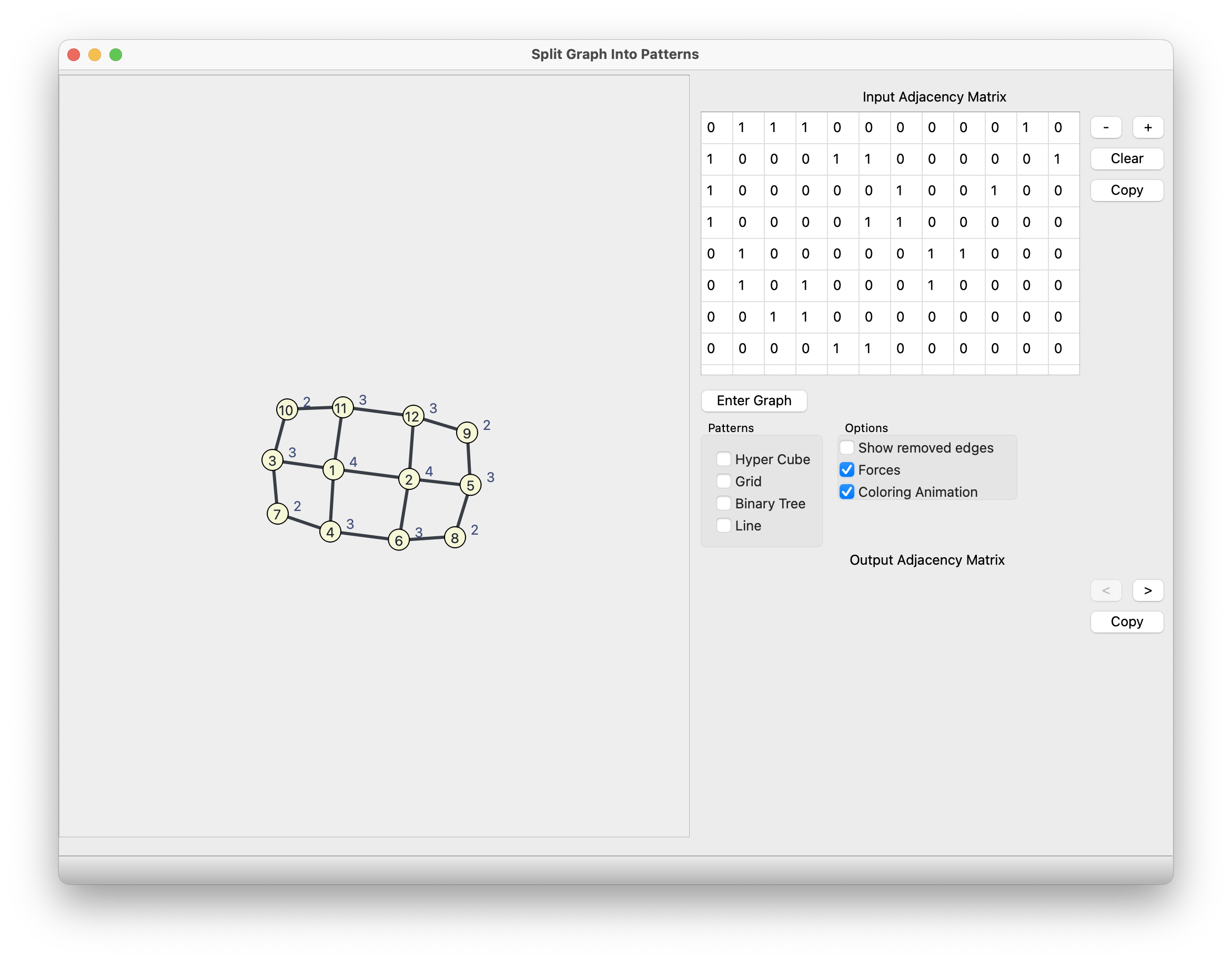
Пусть матрица смежности, описывающая исходны граф является

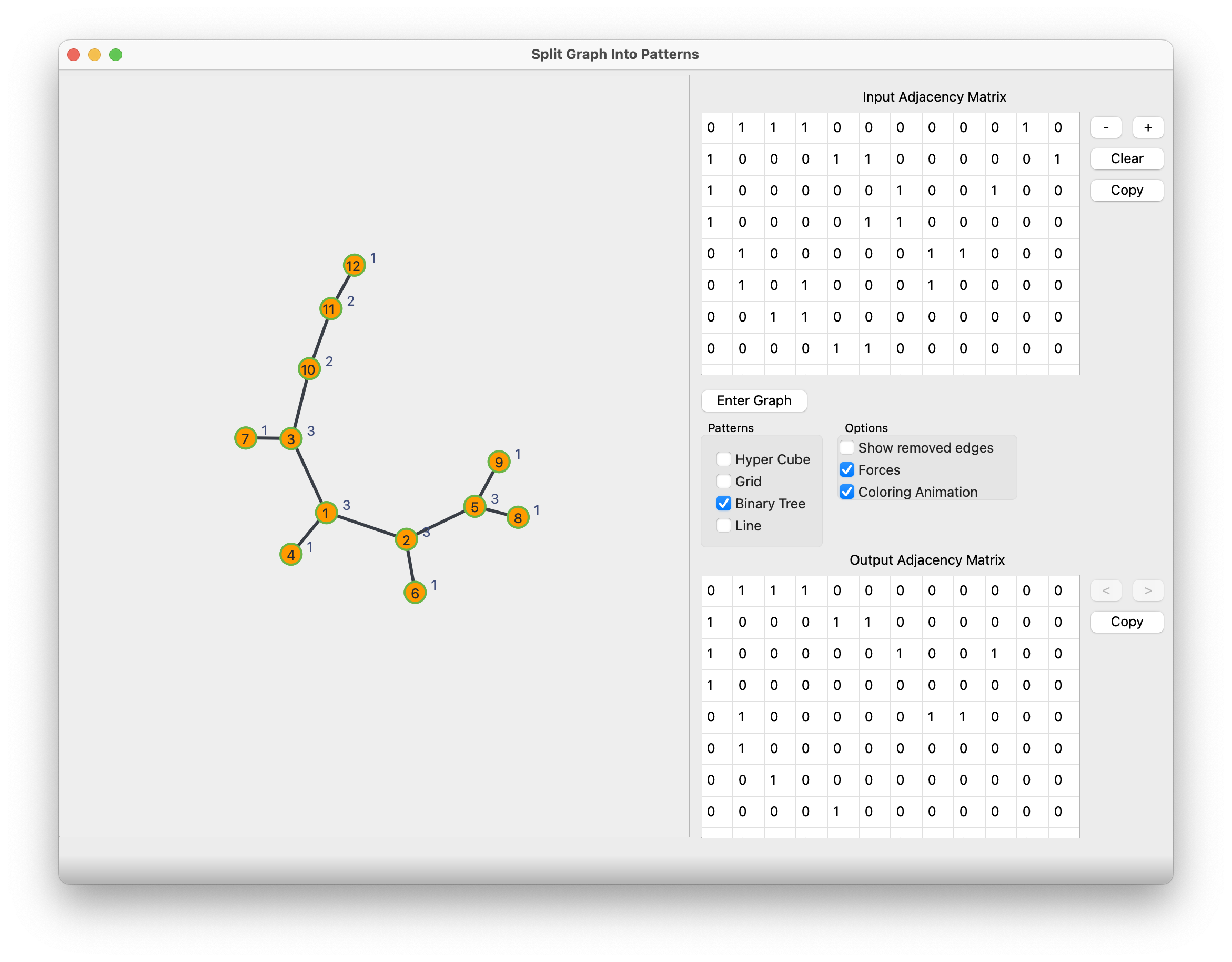
Заполним таблицу для входных данных и нажмем кнопку «Enter Graph». Тогда граф будет иметь вид:

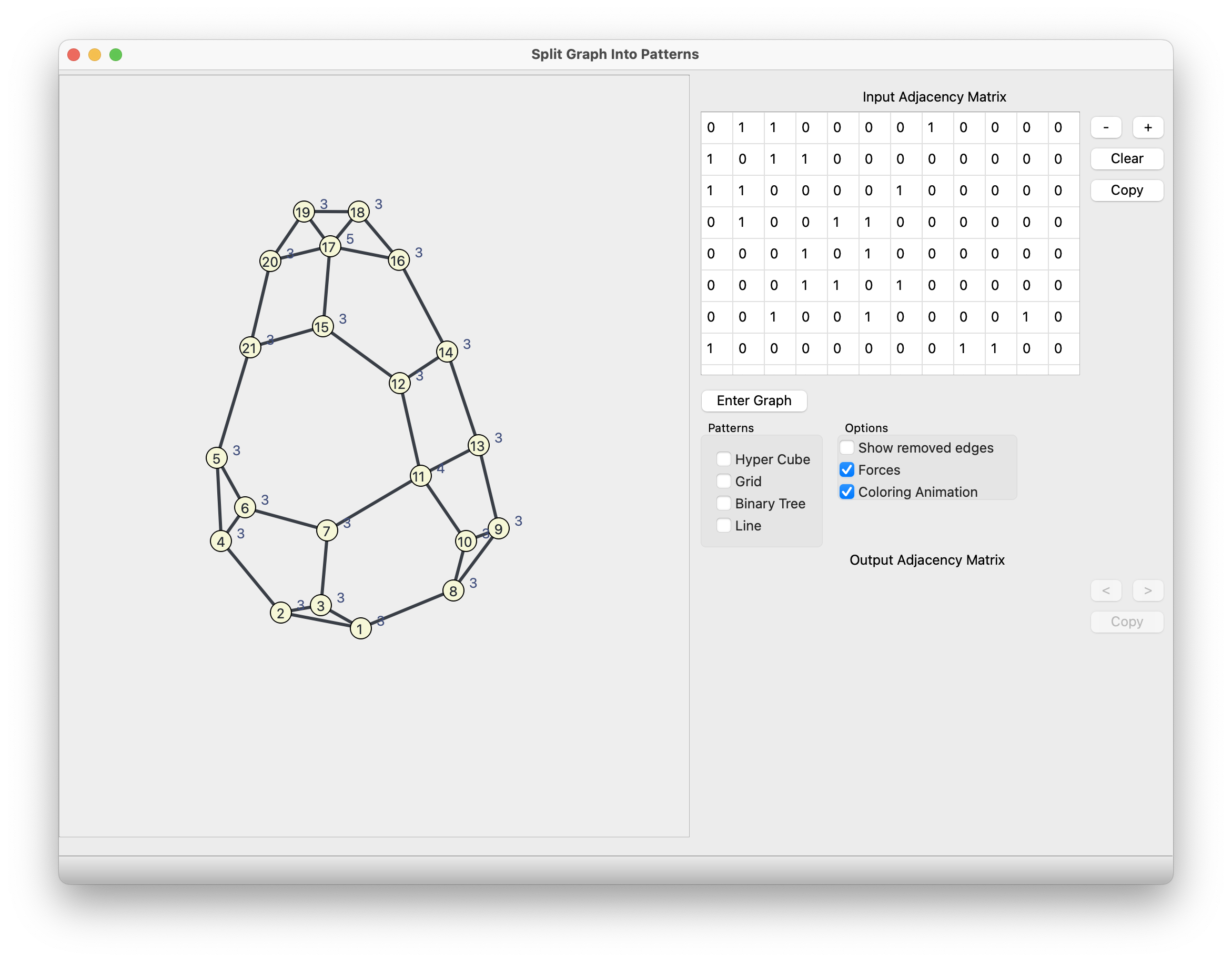
Выделим из него гиперкуб. Для этого необходимо нажать на CheckBox «Hyper Cube». Получим:

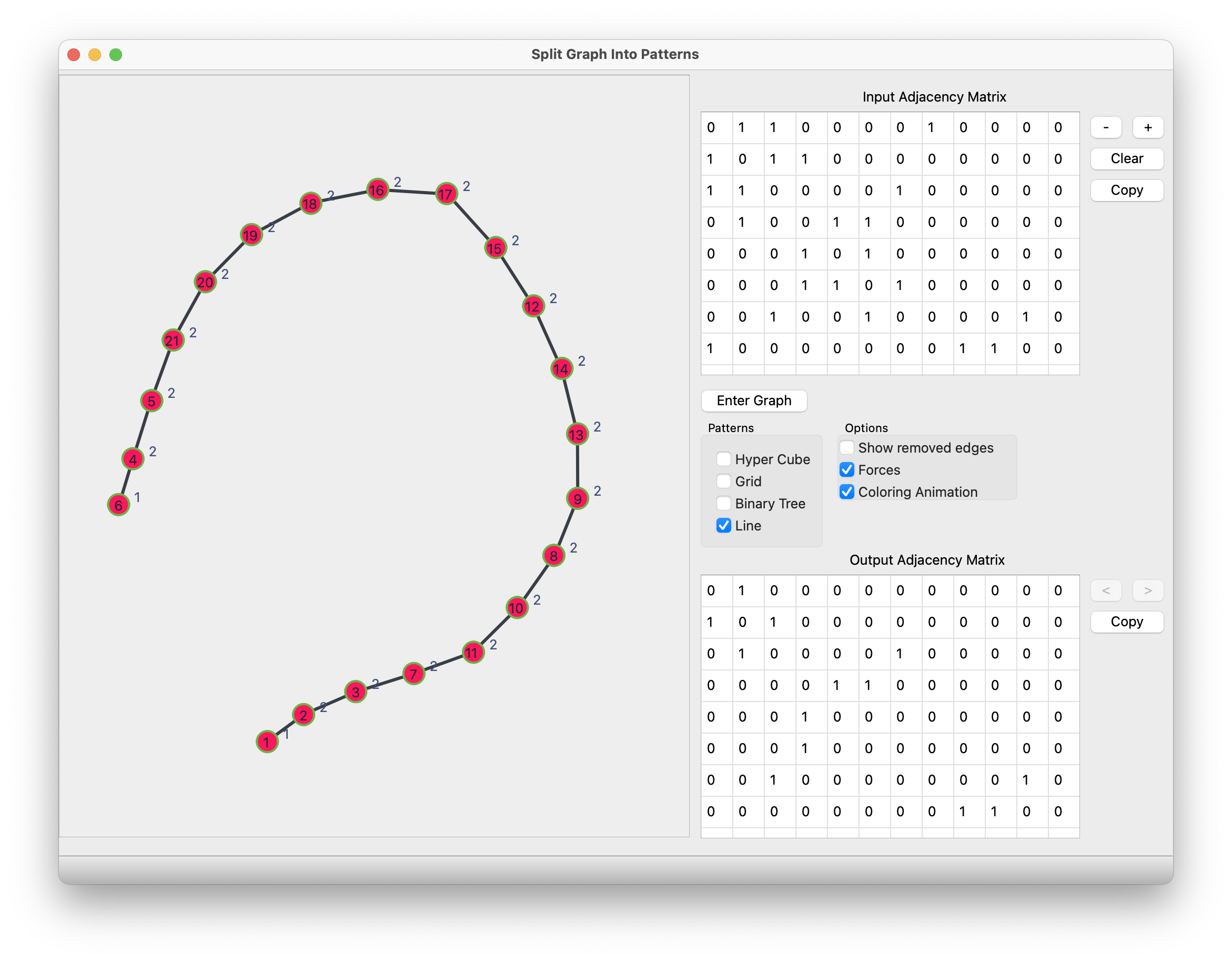
В нижней таблице мы можем видеть матрицу смежности всех компонент связности полученного несвязного графа. Используя кнопки < и >, мы тоже переключаться между ними.

Теперь выделим решетку.

Теперь скопируем полученную матрицу и вставим ее в таблицу для входных данных. Получим

Выделяя бинарное дерево, получим:

Теперь введем другой граф и выделим в нем линию.



Для разработки использовался язык программирования C++ и фреймворк Qt 6.4. Тестирование проводилось на ОС macOS Ventura 13.0. Для работы на других ОС необходима отдельная сборка приложения, используя необходимую ОС.

**Выводы**

В результате выполнения курсовой работы было разработано приложение, позволяющее преобразовывать произвольный граф в лес самоподобных изоморфных графов, а также способное визуализировать полученный результат.